1. Exercise 3.1 in the Lecture 11 handout.

2. Exercise 3.2 in the Lecture 11 handout.

You need to think carefully about this problem. Here are some sequences of A’s and B’s, all having limiting frequencies for the symbol “A”:

<table>
<thead>
<tr>
<th>sequence</th>
<th>limiting frequency of A’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, A, B, A, B, ...</td>
<td>1/2</td>
</tr>
<tr>
<td>A, B, B, A, B, A, B, A, ...</td>
<td>1/3</td>
</tr>
<tr>
<td>A, B, A, B, A, B, A, B, A, B, ...</td>
<td>2/5</td>
</tr>
<tr>
<td>A, B, A, B, A, B, B, A, B, B, A, ...</td>
<td>0</td>
</tr>
</tbody>
</table>

The problem is asking you to produce a sequence (and it can be a sequence consisting of just A’s and B’s) for which there is no limiting frequency for the entry “A”. You might think this can’t be done. But it can, and it’s not really all that hard. Once you get the idea, it’s very simple.

3. Consider the Move-To-Front algorithm applied to a sequence of 4 elements whose initial state is \{A, B, C, D\}. Suppose the sequence of accesses is as follows:


   • Show that the cost using Move-To-Front with this sequence of accesses is 56, and show the final state of the list.
   • What can you say (based on the Sleator-Tarjan theorem) about the cost of the best possible algorithm applied in the same situation?
   • Could there actually be an algorithm that was that good?

4. (a) Read Section 10.1. This should be very easy, since I assume you already know about stacks and queues. The text has an implementation of a queue in terms of an array of fixed size, in which the operations Enqueue and Dequeue each have cost \(O(1)\). (Remember that this is just a fancy way of saying that the cost is uniformly bounded by some constant.) You don’t have to write anything—just read this and understand it. Let me know if you find anything confusing.

   (b) The only problem with that implementation of queues is that the size of a queue is bounded by the size of the array. It would be nicer to have a queue which, like a stack, had no upper bound on its size. (Now of course in practice, a stack does have an upper bound on its size—that’s why we talk about “stack overflow”. But also in practice we have ways of
dealing with this so that except in very unusual circumstances, stacks act as if their size was not constrained.)

So assuming that we had an implementation for stacks in which their size was not constrained, and in which the operations \texttt{Push} and \texttt{Pop} each have cost 1, I first want you to do Exercise 10.1-6 (page 236). The idea of this exercise is to provide an implementation of a queue in which the size of the queue is not constrained. In particular, you should derive the worst-case costs of \texttt{ENQUEUE} and \texttt{DEQUEUE}, and it should be clear that the cost of \texttt{DEQUEUE} is not $O(1)$. Presumably, that’s the price we pay for having a queue of unlimited size.

(c) Then finally, do Exercise 17.3-6 (page 463). The idea of this exercise is show that in the implementation of the queue in Exercise 10.1-6 (which you just did), the amortized cost of both \texttt{ENQUEUE} and \texttt{DEQUEUE} for this data structure is $O(1)$. So things really are not so bad—in fact, they’re pretty good.

5. (From Parberry and Gasarch). An \textit{undirected graph} is a set $V = \{V_1, V_2, \ldots, V_n\}$ of vertices and $E$ of edges (each edge joining two vertices). Any pair of vertices has at most one edge joining them.

A \textit{Hamiltonian cycle} on an directed graph is a path $V_{i_1}, V_{i_2}, \ldots, V_{i_n}$ where

- each consecutive pair $V_{i_k}, V_{i_{k+1}}$ in the path is connected by an edge,
- each vertex in the graph occurs exactly once on the path, and
- There is an edge from $V_{i_n}$ to $V_{i_1}$. (This is what makes it a cycle.)

An undirected graph may or may not have a Hamiltonian cycle.

A \textit{weighted undirected graph} is an undirected graph in which each edge has a weight, which is a positive number that you can think of as the distance between those two vertices. A path (and in particular, a cycle) in a weighted undirected graph has a weight, which is simply the sum of the weights of the edges in the path.

We can model the \textit{traveling salesperson problem} as a problem on a weighted undirected graph—just think of the vertices as cities and the edges as connections (e.g., by plane or car) from one city to another. The problem is to find (if it exists) a Hamiltonian cycle of least weight.

(a) Give an example of an undirected graph having no Hamiltonian cycle.

(b) Here is a naive greedy algorithm that tries to solve the traveling salesperson problem:

\begin{center}
\texttt{Start at vertex number 1, and move along the edge of minimum weight that leads to an unvisited vertex, repeating this until the algorithm arrives at a vertex from which all edges lead to visited vertices. At that point finish up by picking the last edge however you want.}
\end{center}

Is it possible that this algorithm could fail to find a Hamiltonian cycle in the graph, even though there is one?

(c) Suppose the algorithm does find a Hamiltonian cycle. Is it necessarily one of least weight?

Please note that if you assert something is true, you have to prove it. If you assert that it is not true, you have to give a counter-example. Please write clearly!

6. Let $T$ be a rooted tree. And I’m not assuming that $T$ is necessarily a binary tree. That is, each node may have any finite number of children.

We are going to consider a method of visiting each node of the tree. The method works like this:

At Step 1, we visit the root. The root is then marked as “visited”.

At Step $n$, all nodes that have already been marked “visited” can visit at most one of their children. Of course if all the children of a node have already been visited, there is nothing for
that node to do. But other nodes may still do something at that step. (And so in particular, more than one node might be visited in a single step of the process.)

The question is this: what is the minimum number of steps necessary to visit every node in the tree?

(a) Draw a picture of a tree in which more than one node is visited at some step of this process, and show why that happens.

(b) Show that this problem contains optimal substructure. Please be careful when you do this. You have to state precisely and clearly what the substructure is and show why it is optimal.

(c) Use that result to write a recursive algorithm to solve this problem.

(d) Turn that algorithm into a dynamic programming algorithm.

(e) What is the cost of the algorithm? (By that, I mean, “What is the cost of the algorithm that computes the minimum cost?”), not “What is the minimum cost?”.

In doing this problem, please write as little pseudo-code as possible. I would much rather read something that is clearly explained using ordinary language.